# Animation button in android studio

Animations can add visual cues to alert users to what's happening in your app. They are especially useful when changing the UI state, such as when new content loads or new actions are available. Animations also give your app a polished look that gives it a better look and look. Android has different animation APIs depending on the type of animation you want, so this page provides an overview of the different ways you can add motion to the UI. To better understand when you should use animations, also see the material design guide for motion. Image 1 bitmap animation. Animated drawable If you want to animate bitmap graphics such as an icon or illustration, you should use the animation drawing API. Typically, these animations are defined statically with a drawable resource, but you can also define the behavior of the animation at runtime. For example, animating a play button that changes to a pause button when you click it is a nice way to tell the user that the two actions are related and that pressing one of them makes the other visible. For more information, see Animate Drawable Graphics. Visibility to animate the UI and move Figure 2. Subtle animations, when a dialog box appears and disappears, causing the UI to become less abrupt when you need to change the visibility or location of the view in the layout, you should include fine animations to help the user understand how the UI is changing. To move, reveal, or hide the view in the current layout, you can use the property animation system provided by the android.animation package, available in Android 3.0 (API level 11) and above. These APIs update the properties of display objects over a period of time, continuously redrawing the view as property changes. For example, if you change position properties, the view moves across the screen, or when you change the alpha property, the view becomes mymit or minglys. To create these animations with the least effort, you can enable animations in the layout so that when you simply change the visibility of the view, the animation is automatically applied. For more information, see Automatic animation of layout updates. To learn how to create animations using the property animation system, read the property animation overview. Or look at the following pages to create common animations: Physical Motion Figure 3. Animation created using objectanimator figure 4. Animations created using physics-based APIs whenever possible, your animations should use real-world physics to look natural. For example, they should maintain momentum when changing the target and make smooth transitions during any changes. To provide these behaviors, the Android Support Library contains animation APIs physics that rely on the laws of physics to control how your animations occur. Two common physics-based animations are as follows: Spring Animation Fling Fling Animations that are not physics-based—for example, those created using the ObjectAnimator API—are fairly static and have a fixed duration. If the target value changes, you must cancel the animation at the time the target value is changed, reconfigure the animation with the new value as the new starting value, and add a new target value. Visually, this process creates a sudden stop animation and disjoint motion then, as shown in Figure 3. Because animations created using physics-based animation APIs such as dynamicanimation are controlled by force. Changing the target value results in a valid change. The new force applies to an existing speed that allows you to continuously transition to a new target. This process results in a more natural animation, as shown in Figure 4. Change the Layout Animate Image 5. Animations to display additional details can be achieved by changing the layout or by running a new activity on Android 4.4 (API level 19) and above, you can use the transition framework to create an animation when you change the layout within the current activity or fragment. All you have to do is specify the start and end layouts and what type of animation you want to use. Then the system detects and performs an animation between the two layouts. You can use it to mouse the entire UI or move/replace only some views. For example, when a user clicks an item to see more information, you can replace the layout with item details and apply a gradient such as the gradient shown in Figure 5. The start and end layouts are stored in the scene, although the start scene is usually determined automatically from the current layout. You then create a transition to tell the system what type of animation you want, and then call TransitionManager.go() and the system starts the animation to replace the layout. For more information, see Animate between layouts using a gradient. And sample code, check out BasicTransition. Animate between activities on Android 5.0 (API level 21) and above, you can also create animations that move between your activities. This is based on the same transition interface described above to animate layout changes, but allows you to create animations between layouts in separate activities. You can use simple animations, such as moving a new activity from the side or fading it, but you can also create animations that move between shared views in each activity. For example, when a user taps an item to see more information, you can convert it to a new animation activity that seamlessly enlarges the item to fill the screen, such as the animation shown in Figure 5. As usual, you call startActivity(), but give it a package of options provided by ActivityOptions.makeSceneTransitionAnimation(). This option package can include which views are shared between activities, the transition frame can link them during the animation. For all the details, see Start an animation activity. And sample code, check out ActivitySceneTransitionBasic. In Java development, Mobile Development Recipes are community-created content. Ibm does not track or approve them. If you find inappropriate content, let us know using Report Abuse. For more information about community content, please refer to our Terms of Use. Some of them are not yet ready for production, but you can enjoy them a lot of fun. I hope you like it. Here they are not in any particular order: Pull-to-refresh is a great place for creativity! But Yalantis is building a cool pull-to-refresh called Pull To Make Soup animation not just for the purpose of self-expression. These small components can actually help application publishers make their applications stand out. SpaceTabLayout animation that is an amazing TabLayout for Android with a central floating action button. This is a custom relativelayout implementation that you can use along with ViewPager to navigate between fragments. Android Navigation Animation is the main key of any Android app and previously we see an article about animation sidebar, two panel layout animations, bubble tab animations and many more. Consistent navigation is an essential part of the overall user experience. Few things frustrate users more than basic navigation, which behaves in inconsistent and unexpected ways. I assume you are aware of google trends, which is a public web device of Google Inc., based on Google Search, which shows how often a particular search-term is entered in proportion to total search-volume in different regions of the world, and in different languages. Previously, we discuss how to implement a basic animation sidebar in Android and now we are increasing our level by adding Wave Sidebar Animation in the Android app. Most of the time we have to keep the user busy when they perform a click button and work in the background process on Android. So we use the progress indicator or dialog, but here you can use the Android Library Progress button, which is the Android Button, which turns into a load progress indicator. Swipe3DRotateView is a class designed to simplify the implementation of 3D Flip Rotation on android based on swipe gestures. It is extended from Framelayout and should accurately contain 2 views as its children. Detects swipe gestures in child view and rotates children 3D based on the swipe direction on their X or Y axis. CleverRoad introduces its new library called AdaptiveTableLayout, which allows you to read, edit, and write CSV files. If you're using an Android device, you can easily use this library to implement all of these actions. As a result, you'll be able to change rows and columns, view the image using a link, and align the data you want. It will certainly help you cope with your tasks faster and your output higher. AdaptiveTableLayout Android library seems like your disposal. The two Android Library Animation layout panels are fully customizable with a drag-gable divider where you can hide and display fragments vertically and horizontally and customize behavior to change orientation. If the library has a slider separating the fragments, you can drag this slider and the fragments resize the views inside. We recently discussed the Bubble Tab Animation int tab, but you like the Materialize tab then you can add Android PagerSlidingTabStrip. It is an interactive indicator for navigating between different ViewPager pages. Interactive paging indicator widget, compatible with viewpager from android support library. That's it. I hope you enjoyed the article! If I didn't mention any other major libraries released in the animation, please let me know in the comments below. Make the list bigger together! This tutorial shows how to animate a reflection button on Android using Android Studio version 2.3. I assume you know how to create an app in Android Studio. If you don't have this experience yet then I would recommend reading the excellent Building Your first app tutorial from Google first. 1) Add a view of the Start button by placing a button in the res/layout/layout/activity_main.xml activity file. &lt;?xml version=1.0 encoding=utf-8?&gt;&lt;android.support.constraint.ConstraintLayout xmlns:android= xmlns:app= xmlns:tools= android:layout_width=match_parent android:layout_height=match_parent tools:context=com.evgenii.sixbouncingbuttons.MainActivity&gt; &lt;Button android:id=@+id/button android:layout_width=92dp android:layout_height=92dp android:onclick=didTapButton android:background=#FFA400 app:layout_constraintbottom_tobottomof=parent app:layout_constraintleft_toleftof=parent app:layout_constraintright_torightof=parent app:layout_constrainttop_totopof=parent&gt;&lt;/Button&gt; &lt;/android.support.constraint.ConstraintLayout&gt; 2) Vytvořte animaci měřítka Dále vytvoříme soubor animace res/anim/bounce.xml pro změnu měřítka zobrazení. Right-click the res folder. Select New Resource File / Android Resource File. Type bounce as the file name Choose the animation resource type. The directory name field changes to anim. Next, open the res/anim/bounce.xml file that was created for you and replace its contents with the following code.&lt;?xml version=1.0 encoding=utf-8?&gt;&lt;set xmlns:android= amp;gt; &lt;scale android:duration=2000 android:fromxscale=0.3 android:toxscale=1.0 android:fromyscale=0.3 android:toyscale=1.0 android:pivotx=50% android:pivots=50%&gt;&lt;/scale&gt; &lt;/set&gt; This code creates animation, changes changes from 30% to 100% in two seconds. 3) Respond to click now we add code that animes the button on tap. Add the following method to the Java activity file. public void didTapButton(View view) { Button button = (Button)findViewById(R.id.button); final animation myAnim = AnimationUtils.loadAnimation(this, R.anim.bounce); button.startAnimation(myAnim); } When you launch an app and click a button, it will effectively animate from smaller to larger in size. 4) Implement bounce interpolator Further, we write code that adds a reflection effect to the animation scale. Create a new Java class file in your application module and name it MyBounceInterpolator. Open the Java file that was created and replace the class code with the following. MyBounceInterpolator class implements android.view.animation.Interpolator { private double mAmplitude = 1; private double mFrequency = 10; MyBounceInterpolator(double amplitude, double frequency) { mAmplitude = amplitude; mFrequency = frequency; } public float getInterpolation(float time) { return (float) (-1 * Math.pow(Math.E, -time/ mAmplitude) * Math.cos(mFrequency * time) + 1); } } I'll explain how this code works at the moment. 5) Use bounce interpolator finally, open your Java activity file again and replace the entire didTapButton method with the following code. public void didTapButton(View view) { Button button = (Button)findViewById(R.id.button); final animation myAnim = AnimationUtils.loadAnimation(this, R.anim.bounce); // Use a bounce interpolator with an amplitude of 0.2 and a frequency of 20 MyBounceInterpolator interpolator = new MyBounceInterpolator(0.2, 20); myAnim.setInterpolator(interpolator); button.startAnimation(myAnim); } Here we have added MyBounceInterpolator to the animation with the setInterpolator method. When you launch an app and click a button, it animates with a spring effect. How the bounce animation interpolator works We have initialized the MyBounceInterpolator object with two arguments. MyBounceInterpolator interpolator = new MyBounceInterpolator(0.2, 20); The first value of 0.2 is the reflection amplitude. A higher value produces more pronounced reflections. The second value of 20 is the frequency of bounces. A higher value creates more woes during the animation time period. To achieve the reflection effect, the getInterpolation method maps time using the following function: In this equation there are a and w amplitute and frequency values, and t is the time. The equation contains a cosine function that causes periodic fluctuations during animation. To reduce its amplitute with time, we multiply the cosine by exponential function. The following chart shows that the animation exceeds 1 first and then settles closer to it. Link Posted May 30, 2016 by Evgenii 2016 by Evgenii